```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 22 10:10:26 2019

@author: Nate



1.) Compute the exact ground state energy of the Helium atom by use of the Diffusion
Monte Carlo algorithm with importance sampling. Use the trial function introduced
in problem 3 of HW7. Read Moskowitz et al, J. Chem. Phys. 77(1982)349, S. A.
Chin, Phys. Rev. 42(1990)6991. Not all the materials in these two paper are equally
relevant.

The basic idea of DMC is to use the Langevin algorithm to iterate the configurations
of the system (the two electrons' position in the case of Helium), but ADDITION-
ALLY, replicated each configuration according to the exponential of the local energy
as described in the lecture note. When computing the energy expectation values,
average over ALL configurations (include those replicated ones).

"""

import numpy as np
import matplotlib.pyplot as plt
import pdb
import Metropolis_Module as mm
import random

###############################################################################
###############################################################################

#define functions
#advance position by dt using langevine algorithm: x' = x+vdt+g*t^.5
#Langevin algorithm
def langevin_alg(x, v, dt):
    x = x + v*dt + np.sqrt(dt)*np.random.randn()
    return x

#evaluate ratio
def evaluate_ratio(r1, r1_trial,r2, r2_trial, alpha, dt):
    global E_trial


    #gb = np.exp(-.5*((energy_calc(r1, r2, alpha) + energy_calc(r1_trial, r2_trial, alpha)) - E_trial)*dt)
    #gb2 = np.exp(-.5*((energy_calc(r1, r2, alpha) + energy_calc(r1_trial, r2_trial, alpha)) - E_trial)*dt)

    gd = np.exp(-(np.linalg.norm(r1 - r1_trial + dt*alpha*r1_trial/np.linalg.norm(r1_trial)))/(2*dt))
    gd2 = np.exp(-(np.linalg.norm(r2 - r2_trial + dt*alpha*r2_trial/np.linalg.norm(r2_trial)))/(2*dt))

    gd_inv = np.exp(-(np.linalg.norm(r1_trial - r1 + dt*alpha*r1/np.linalg.norm(r1)))/(2*dt))
    gd2_inv = np.exp(-(np.linalg.norm(r2_trial - r2 + dt*alpha*r2/np.linalg.norm(r2)))/(2*dt))

    A = np.exp(-2*alpha*np.linalg.norm(r1))*np.exp(-2*alpha*np.linalg.norm(r2))*gd_inv*gd2_inv
    B = np.exp(-2*alpha*np.linalg.norm(r1_trial))*np.exp(-2*alpha*np.linalg.norm(r2_trial))*gd*gd2
    ratio = B/A

```

```
    if ratio > np.random.uniform():
        return(True)
    else:
        return(False)


#Calculate energy
def energy_calc(r1, r2, alpha):
    r1 = np.array(r1)
    r2 = np.array(r2)

    r1mag = np.linalg.norm(r1)
    r2mag = np.linalg.norm(r2)
    r12mag = np.linalg.norm(r2-r1)

    E1 = -(1/2.0)*alpha**2 + alpha/r1mag-2/r1mag
    E2 = -(1/2.0)*alpha**2 + alpha/r2mag-2/r2mag
    energy = E1 + E2 + (1.0/r12mag)

    return energy




################################################################################
################################################################################
#Initial Conditions
N = 500
alpha = 1.6875
E_theory = -2.9073
#dt_list = [0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1]
dt_list = [i*.003 for i in range(1,26)]
#dt = .01
param = .01
before_eval_loop_number = 1
after_eval_loop_number = 1000
E_dt_avg_vs_dt = []
count=0
####Initialize pairs of particles, [x1,y1,z1,x2,y2,z2] using metropolis algorithm
#Initialise an ensemble of $N_{c}$ configurations, which should be uncorrelated and
distributed according
#to the probability density of the guiding function $\vert\Psi_G\vert^2$.


r_0 = [[[1,2,3],[4,5,6]]]
#generate x,y,z,x2,y2,z2

#loop through metropolis 10000 times generating new points
for i in range(10000):
    r_0.append(mm.metropolis(r_0[-1][0], r_0[-1][1], alpha))



for dt in dt_list:
    #take last 500 and append to r
    r = []
    for i in range(N):
        r.append(r_0[-i*5])


    #Initialise the trial energy $E_{T}$ to the average VMC energy of the ensemble.
    E_trail = 0

    for pair_index in r:
        E_trail += energy_calc(pair_index[0], pair_index[1], alpha)

    E_trail = E_trail/N
```

```
    
    
    
    ##########################################################################
    ##########################################################################
    #main loop
    
    E_dt = []
    
    for l in range(after_eval_loop_number):
    
        #loop 1e2 ~ 1e3 to update E_T
        for j in range(before_eval_loop_number):
            r_new = []
        #propose move via langevin_alg and accept or reject via Metropolis probability
            for pair_index in range(len(r)):
                current_pair = r[pair_index]
                r1 = np.array(current_pair[0])
                r2 = np.array(current_pair[1])
                r1mag = np.linalg.norm(r1)
                r2mag = np.linalg.norm(r2)
    
                r1_trial = [0.0,0.0,0.0]
                r2_trial = [0.0,0.0,0.0]
    
                for i in range(len(r1)):
                    r1_trial[i] = langevin_alg(r1[i], -alpha*r1[i]/r1mag, dt)
                for i in range(len(r2)):
                    r2_trial[i] = langevin_alg(r2[i], -alpha*r2[i]/r2mag, dt)
    
                r1_trial = np.array(r1_trial)
                r1_trial_mag = np.linalg.norm(r1_trial)
                r2_trial = np.array(r2_trial)
                r2_trial_mag = np.linalg.norm(r2_trial)
    
    
                #accept or reject the move
                #calculate branching factor
                #multiply number of walkers
                if evaluate_ratio(r1, r1_trial,r2, r2_trial, alpha, dt):
                    p = np.exp(-dt * (.5*(energy_calc(r1_trial, r2_trial, alpha) +
                    energy_calc(r1, r2, alpha)) - E_trail))
                    if abs(p-int(p)) > np.random.uniform():
                        mult = int(p)+1
                    else:
                        mult = int(p)
                    for i in range(mult):
                        r_new.append([list(r1_trial), list(r2_trial)])
    
                else:
                    p = np.exp(-dt * (energy_calc(r1, r2, alpha) - E_trail))
                    if abs(p-int(p)) > np.random.uniform():
                        mult = int(p)+1
                    else:
                        mult = int(p)
                    for i in range(mult):
                        r_new.append([list(r1), list(r2)])
    
            #print(len(r),len(r_new))
            r = r_new
    
    
        #update E_t
        E_trail = 0
        for pair_index in r:
            E_trail += energy_calc(pair_index[0], pair_index[1], alpha)
        E_trail = E_trail/len(r)
```

```
                E_trail = E_trail - param/dt*np.log(len(r)/N)

                #renomalize number of walkers
                while len(r) > N:
                    r.remove(random.choice(r))
                while len(r) < N:
                    r.append(random.choice(r))

                E_dt.append(E_trail)




            ugh = 700
            E_dt_avg = []
            for i in range(len(E_dt)):
                if i >= ugh:
                    E_dt_avg.append(np.average(E_dt[ugh:i+1]))

            E_dt_avg_vs_dt.append(E_dt_avg[-1])

            count +=1
            print("finished: ",dt, E_dt_avg_vs_dt[-1], count)

    print(E_dt_avg_vs_dt)
    ##############################################################################
    ##############################################################################
    #Plotting

    fig1, axes1 = plt.subplots()
    axes1.scatter(dt_list, E_dt_avg_vs_dt, label = 'Calcuated Energy')
    axes1.plot(dt_list, [-2.8477 for i in range(len(dt_list))], linestyle='dashed', label =
    'Theoretical')
    axes1.plot(dt_list, [-2.9073 for i in range(len(dt_list))], linestyle='dashed', label =
    'Theoretical')
    axes1.set_ylabel('Energy Average')
    axes1.set_xlabel('Time Step Size $\Delta t$')
    axes1.set_title("Energy Average vs Time Step Size $\Delta t$", va='bottom')
    plt.show()
```